

# Module 3: Using PkgTemplates to Create Scientific Julia Packages

Madhusudhan Pandey, PhD

December 2025

## Abstract

This module provides a practical, workflow-oriented lecture on how I use `PkgTemplates.jl` to generate professional, reproducible scientific Julia packages. While Module 1 introduced the overall repository structure used in modern scientific packages (such as `ModelingToolkit.jl`), this module shifts focus to the real workflow of *creating* a `ModelingToolkit`-based package from scratch.

I present a first-person narrative of how I configure templates, generate package skeletons, add modeling functionality, create unit tests, and integrate continuous integration workflows. Using `PendulumSystem.jl` as an example, this module demonstrates how I move from high-level modeling goals to a fully operational Julia package with minimal boilerplate. This document is part of a multi-module series. This is **Lecture Module 3**. Each module in this lecture series is designed as a self-contained 30-minute session. The material may be taken independently or with an instructor, and is suitable for learners at the beginner, intermediate, or advanced level. This flexibility allows students to progress at their own pace while ensuring that each module provides a complete and accessible learning experience.

## 1 Introduction

Whenever I start a new scientific modeling project in Julia, I immediately ask myself a set of practical questions: Where will my model code live? How do I structure tests? How do I ensure that the package remains maintainable as the project grows?

Before I used `PkgTemplates.jl`, I manually created the same folders repeatedly: `src/`, `test/`, `Project.toml`, `LICENSE`, and `README.md`. After doing this many times, I realised that writing scaffolding was taking time away from scientific modeling.

`PkgTemplates.jl` solved this problem. It generates a complete, professional Julia package layout in a single command, ensuring consistency, reproducibility, and compliance with the SciML/`ModelingToolkit` ecosystem’s best practices. By automating the “boilerplate” steps, I can focus on designing models rather than setting up directories.

## 2 Why I Use PkgTemplates

I think of `PkgTemplates` as a *package generator*. It creates:

- a clean repository structure,
- a correct `Project.toml`,
- a `src/PackageName.jl` file,
- optional Git initialisation,
- optional GitHub Actions CI workflows,

- optional documentation folders,
- unit tests,
- licenses and formatter configurations.

This matches the repository organisation described in Module 1 for ModelingToolkit.jl<sup>1</sup>.

Whenever I begin work on a new scientific model, such as a pendulum or a hydropower subsystem, `PkgTemplates` becomes my starting point. It encodes my workflow so that every new project begins with the same high-quality layout.

### 3 Installing and Loading PkgTemplates

Installation is straightforward:

```
using Pkg
Pkg.add("PkgTemplates")
```

Once installed, I load it using:

```
using PkgTemplates
```

At this point, I am ready to define a reusable template configuration.

### 4 Creating My Template Configuration

I design my template based on the long-term lifecycle of the project. Below is a typical configuration:

```
t = Template(
    user="pandeyesudan1",
    authors="Madhusudhan Pandey",
    plugins=[
        Git(),
        GitHubActions(),
        License(; name="MIT"),
        Tests(),
        SrcDir(),
        GitIgnore()
    ]
)
```

Each plugin serves a specific purpose:

- **Git:** Enables immediate version control.
- **GitHubActions:** Provides continuous integration (CI).
- **License:** Ensures open-source accessibility.
- **Tests:** Adds a unit testing framework.
- **SrcDir:** Creates a `src/` directory with a module stub.
- **GitIgnore:** Keeps the repository clean.

This template captures my “default” scientific workflow.

---

<sup>1</sup>See Module 1 PDF for the repository structure overview.

## 5 Generating a ModelingToolkit Package

To create the package `PendulumSystem.jl`, I run:

```
t("PendulumSystem")
```

This generates:

```
PendulumSystem/  
  Project.toml  
  src/  
    PendulumSystem.jl  
  test/  
    runtests.jl  
  README.md  
  LICENSE  
  .gitignore  
  .github/workflows/CI.yml
```

The structure mirrors the professional organisation shown in Module 1, including testing, documentation, CI automation, and clean separation of code.

## 6 Adding ModelingToolkit

Inside the new package directory, I add ModelingToolkit as a dependency:

```
cd("PendulumSystem")  
using Pkg  
Pkg.add("ModelingToolkit")
```

This updates `Project.toml` to include ModelingToolkit in the `[deps]` section. Now the package is ready to construct symbolic systems.

## 7 Writing the First Scientific Model

I open `src/PendulumSystem.jl` and replace the default content with a simple pendulum model. The function `pendulum_system()` returns a ModelingToolkit `ODESystem`, allowing any user to simulate pendulum dynamics with a single function call.

This is the first meaningful scientific component of the package.

## 8 Adding Basic Tests

I add a minimal test in `test/runtests.jl`:

- Does `pendulum_system()` return an `ODESystem`?
- Does the package load successfully?

This small test is enough to ensure basic correctness and acts as a foundation for future, more detailed test suites.

## 9 Running Tests and Pushing to GitHub

Once the tests pass, I create the initial commit and push the package to GitHub. Because the template included `GitHubActions`, my tests automatically run on multiple Julia versions and operating systems.

This confirms the package is stable, CI-ready, and compliant with scientific software expectations in the SciML ecosystem.

## 10 Conclusion

This module demonstrated how I use `PkgTemplates.jl` to rapidly generate scientific Julia packages with professional structure, automated testing, and integrated continuous integration. Using `PendulumSystem.jl` as an example, I showed how model development becomes smoother and more reproducible once the boilerplate is automated.

In Module 4, I will extend this workflow further by combining `PkgTemplates` with `ModelingToolkit`'s component-based modeling and connector systems to build multi-domain scientific models.